

# RoBuSt: A Crash-Failure-Resistant Distributed Storage System\*

Revised full version

Martina Eikel

Christian Scheideler

Alexander Setzer

University Paderborn, Germany

In this work we present the first distributed storage system that is provably robust against crash failures issued by an adaptive adversary, i.e., for each batch of requests the adversary can decide based on the entire system state which servers will be unavailable for that batch of requests. Despite up to  $\gamma n^{1/\log \log n}$  crashed servers, with  $\gamma > 0$  constant and  $n$  denoting the number of servers, our system can correctly process any batch of lookup and write requests (with at most a polylogarithmic number of requests issued at each non-crashed server) in at most a polylogarithmic number of communication rounds, with at most polylogarithmic time and work at each server and only a logarithmic storage overhead.

Our system is based on previous work by Eikel and Scheideler (SPAA 2013), who presented IRIS, a distributed information system that is provably robust against the same kind of crash failures. However, IRIS is only able to serve lookup requests. Handling both lookup and write requests has turned out to require major changes in the design of IRIS.

## 1 Introduction

One of the main challenges of a distributed system is that it is able to work correctly even if parts of the system fail to work. If a server experiences a *crash failure* it becomes unavailable to the other servers, i.e., it does not issue or respond to requests any more. Crash failures can be temporary or permanent, and if it is temporary, a server may either be back to its state when it crashed, or it may have lost all of its state. We will focus on crash failures where, whenever a server becomes available again, it is back to its state when it crashed. This is a reasonable assumption since for commercial servers it is extremely rare that their state cannot be recovered. However, a temporary

---

\*This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901) and by the EU within FET project MULTIPLEX under contract no. 317532.

unavailability is not that uncommon and can have many causes such as maintenance work, hardware or software glitches, or denial-of-service attacks. Especially denial-of-service attacks can be a serious threat because they are normally unpredictable, hard to prevent and they can cause the unavailability of a server for an extended period of time.

Predominant approaches in information and storage systems to deal with the threat of crash failures are to use redundancy: information that is replicated among multiple machines is likely to remain accessible even if some servers are unavailable. Unfortunately, in systems that consist of thousands of servers a complete replication of the data over all servers is not feasible. Hence, one needs to find an appropriate tradeoff between the amount of redundancy and the number of crashed servers the system can handle. One can easily show that if  $\Theta(\log n)$  copies of a data item are placed randomly among  $n$  servers, and these random positions are not known to the adversary, then any strategy of the adversary that blocks half of the servers will not block all of the copies, with high probability<sup>1</sup>. The situation is completely different, however, when considering an adaptive adversary, i.e., someone who has complete knowledge about the system.

In a previous work, Eikel and Scheideler [7] presented a distributed information system, called IRIS, that just needs a constant storage redundancy in order to be robust against an adaptive adversary that can crash up to  $\Theta(n^{1/\log \log n})$  servers. Unfortunately, the system lacks the important ability to handle write requests, i.e., to add, remove and update data items. This work solves this problem.

## 1.1 Model and Preliminaries

We assume that the storage system consists of a static set  $\mathcal{S} = \{s_1, \dots, s_n\}$  of  $n$  reliable servers of identical type. The servers are responsible for storing the data as well as handling the user requests. We assume that all data items are of the same size, and that any data item  $d$  is uniquely identified by a key  $key(d)$ . The universe of all possible keys is denoted by  $U$ , and  $m := |U|$  is assumed to be polynomial in  $n$ . Furthermore, we assume that the size of the data items is at least  $\Omega(\log n \log m)$ . There are two types of user requests:  $lookup(k)$  for  $k \in U$ , and  $write(k, d)$  for  $k \in U$  and a data item  $d$ . The user can issue a request by sending it to one of the servers in  $\mathcal{S}$ . Given a  $lookup(k)$  request, the system is supposed to either return the data item  $d$  with  $key(d) = k$ , or to return NULL if no such data item exists. Given a  $write(k, d)$  request, the system is supposed to store data item  $d$  with key  $k$  such that subsequent  $lookup(k)$  requests can be answered correctly. Note that with a  $write(\cdot)$  request the user can also update or remove data.

Every server knows about all other servers and can therefore directly communicate with any one of them. This does not endanger scalability since millions of IP addresses can easily be stored in main memory in any reasonable computer today and we assume the set of servers to be static. We use the standard synchronous message passing model

---

<sup>1</sup>“With high probability”, or short, “w.h.p.”, means a probability of at least  $1 - 1/n^c$  where the constant  $c$  can be made arbitrarily large.

for the communication between the servers. That is, time proceeds in synchronized *communication rounds*, or simply *rounds*, and in each round each server first receives all messages sent to it in the previous round, processes all of them, and then sends out all messages that it wants to send out in this round. Note that assuming the synchronous model is just a simplification and that our protocols only require the message delays to be bounded. In addition, we use the synchronous model because describing all protocols in an asynchronous setting would significantly blow up the construction and would hide the main innovations behind our system. We assume that the time needed for internal computations is negligible, which is reasonable as the operations in the protocols we describe are simple enough to satisfy this property.

For the crash failures, we assume a *batch-based* adaptive adversary. This means the following: We assume that time is divided into *periods* consisting of a polylogarithmic number of rounds. The adversary has complete knowledge of the current system, but cannot predict the (future) random choices of the system. Based on his knowledge, he can select an arbitrary set of  $O(n^{1/\log \log n})$  servers to be crashed. A server that is crashed will not send any message nor react to messages sent from other servers. We assume that the servers have a failure detector that allows them to determine whether a server is crashed so that statements like “if server  $i$  is crashed then ...” are allowed in the protocol. Note that assuming bounded message delays, failure detection can simply be implemented using timeouts. After that, the adversary may issue an arbitrary collection of requests to the system by sending up to  $\omega \in \mathbb{N}$  `lookup( $\cdot$ )` requests and up to  $\omega$  `write( $\cdot$ )` requests to each server. In order to keep the presentation of RoBuSt as clear as possible, throughout this work we assume  $\omega = 1$ . RoBuSt can in principle handle arbitrary values of  $\omega$ , but in that case the bound on the work required by each server for serving all requests must be multiplied with  $\omega$ .<sup>2</sup> There are no further limitations, i.e., the keys selected by the adversary may or may not be associated with data items stored in the system, and the adversary is also allowed to issue multiple lookup requests for the same key. The task of the system is to correctly handle *all* of these requests. We assume that any period is long enough such that the system has enough time to perform all necessary computations and to answer all requests. After any period, the adversary may select a different set of  $\Theta(n^{1/\log \log n})$  servers to be crashed. We assume that the set of crashed servers does not change during a fixed period, which is why we use the notion of a batch-based adaptive adversary. Of course, allowing crash failures at arbitrary times would make the model much stronger, yet it would significantly complicate the system design, which is why we leave this to future research. Note that we assume links between intact (i.e., non-crashed) servers to be reliable. Unreliable links can be dealt with using, for example, at-least-once delivery or error correction strategies, which are out of scope for our design since it is already complex enough.

In order to measure the quality of the storage system, we introduce the following notation. A storage strategy is said to have a *redundancy* of  $r$  if  $r$  times more storage (including any control storage) is used for the data than storing the plain data. We

---

<sup>2</sup>Note that our system would not be able to answer all requests with at most polylogarithmic work if  $\omega > \text{polylog}(n)$ , but this would trivially hold for any storage system.

call a storage system *scalable* if its redundancy is at most  $\text{polylog}(n)$ , *efficient* if any collection of lookup and write requests specified by the adversary can be processed correctly in at most  $\text{polylog}(n)$  many communication rounds in which every server sends and receives at most  $\text{polylog}(n)$  many messages of at most  $\text{polylog}(n)$  size, and *robust* if any collection of lookup and write requests specified by the adversary can be processed correctly even if a set of up to  $\Theta(n^{1/\log \log n})$  servers specified by the adversary crash.

## 1.2 Related Work

Over the past years, distributed storage systems have gained a lot of importance. Popular examples include the storage solutions offered by Google, Apple, or Amazon. Since availability and retrievability of the stored data is a key aspect of distributed storage systems, these systems should be able to work correctly despite common failures. Often failures in distributed systems are divided into the following types [5]: crash failures, omission failures, timing failures, and Byzantine failures. In crash failures the affected component (for instance a server) completely stops working. In receive (send) omission failures the affected component cannot receive (send) any further messages. A timing failure leads a component to not respond within a specified time interval. In case of a Byzantine failure, the affected component may react in an arbitrary, even malicious manner.

This work focuses on crash failures. Many works dealing with crash failures in distributed systems focus on crash failure recovery and crash failure detection [15, 12, 8]. But to the best of our knowledge, no previous work has considered how to secure a distributed storage system against many (e.g., more than a polylogarithmic number) simultaneous crash failures controlled by an adaptive adversary while using only polylogarithmic work, time and redundancy. That is, we do not seek to prevent failures or attacks, but rather focus on how to maintain a good availability and performance even in spite of them. Our system is based on the distributed hash table (DHT) paradigm (e.g., [3, 6, 9, 14, 16]), with the additional twist of using coding and arranging the used DHTs in an appropriate structure. Various systems based on DHTs that are resistant against Denial-of-Service (DoS) attacks (which represent a special type of crash failures) have already been proposed [10, 11, 13]. But these do not work for adaptive adversaries. The first DHTs that are robust against past insider crash failures were proposed in [1, 2], where a past insider only has complete knowledge of the information system up to some *past* time point  $t_0$ . For this kind of insider, it is possible to design an information system so that any information that was inserted or last updated *after*  $t_0$  is safe against crash failures [1, 2]. But the constructions proposed in these papers would not work at all for a current insider because they are heavily based on randomization to ensure unpredictability. Eikel and Scheideler were the first to present a distributed information system, called IRIS, that is provably robust even against a current insider that crashes up to  $\Theta(n^{1/\log \log n})$  servers. The authors showed that IRIS can correctly answer any set of lookup requests (with one request per server that is not crashed) with polylogarithmic time and work at each server and only a constant redundancy. Still it remained open whether it is possible to design a distributed storage system that

can efficiently handle lookup and write requests under the presence of crash failures. We answer this question positively by proposing such a system.

### 1.3 Our Contribution

We present the first scalable distributed storage system, called *Robust Bucket Storage* (in short *RoBuSt*), that is provably robust against adaptive crash failures and that supports both lookup and write requests. Concretely, we allow the adversary to have complete knowledge about the storage system and to have the power to crash any set of  $\gamma n^{1/\log \log n}$  servers, for  $\gamma > 0$  constant. The task of the system is to serve any collection of lookup and write requests in an efficient way despite the crash failures.

RoBuSt expands some of the ideas in IRIS, a distributed storage system that we proposed in SPAA 2013 [7]. The system presented in this work tolerates a number of crashed servers that is similar to the number of servers blocked by a DoS attack that the Basic IRIS version can tolerate and achieves comparable efficiency bounds (up to a logarithmic factor). In contrast to IRIS, which can only handle lookup requests, RoBuSt is able to additionally handle write requests. Although in the lookup protocol we are able to adapt some of the underlying ideas of IRIS, adding the write functionality required significant changes in the whole structure. To simplify the description for readers who are familiar with IRIS, we try to re-use terminology whenever there are similarities (e.g., Probing Stage, Decoding Stage).

One aspect is that IRIS organizes data into layers of  $n$  data items each, and each layer is encoded separately using distributed coding that involves all  $n$  servers. This means that whenever a data item needs an update, all  $n$  servers have to update their information for the corresponding layer. Since we allow any set of write requests, it may happen that every write request involves a different layer, which would create an enormous update work. To solve this issue, in RoBuSt we store the data items in so-called buckets that are organized in a binary tree. For each data item, there are a logarithmic number of buckets that are a potential storage location for the data item. For a data item there may exist different versions of it in different buckets. But our system ensures that the highest bucket (i.e., the bucket with minimum distance to the root in the underlying binary tree over the buckets) that contains a version of the data item always holds the most recent version.

Furthermore, IRIS uses a fixed set of hash functions to specify anchor locations for the data so that afterwards lookup requests can be served efficiently despite an adversarial DoS attack. However, using fixed hash functions in RoBuSt would enable the adversary to annul the fair distribution of data in a bucket. Therefore, RoBuSt chooses new, random hash functions whenever write requests have to be served.

Another complication is the fact that a server may not know whether its information is up-to-date. This is because at the time when write requests were executed that required an update in that server, the server might have been crashed. Our organization of the data and our protocols ensure that any server that answers a request always returns the most recent version of a data item.

Nevertheless, RoBuSt makes sure that all data can still be efficiently found while the storage overhead is at most a logarithmic factor.

**Theorem 1.1.** *RoBuSt is a scalable and efficient distributed storage system that only needs a logarithmic redundancy to protect itself against batch-based adaptive crash failures in which up to  $\gamma \cdot n^{1/\log \log n}$  servers can crash for a constant  $\gamma > 0$ , w.h.p.*

## 2 Underlying Datastructure

In the following, we assume keys are potentially from an address space of size at most  $n^p$ , i.e., we need  $\Lambda := p \log n$  bits for each address. We introduce the following definitions: For a data item  $d$ , denote the *address* of  $d$  by  $\text{key}(d) = d_{p \log n - 1} \dots d_1 d_0 \in \{0, 1\}^{p \log n}$  and let  $\text{bit}_d(i) := d_i$ .

Our data structure is based on a binary tree with  $\Lambda + 1$  levels, so-called *zones*. We denote the nodes of each zone as *buckets* where each bucket will hold a set of data items. The internal storage strategy of the buckets is described in Section 2.1. Zone 0 consists of a single bucket, bucket  $B_\varepsilon$ . Each bucket  $B$  that is not in zone  $\Lambda$  has two children, denoted by  $0\text{-child}(B)$  and  $1\text{-child}(B)$ . For each data item  $d$  there is not only a single possible bucket in which to store  $d$  but there are  $\Lambda + 1$  possible buckets for  $d$ , one in each zone. Bucket  $B_\varepsilon$  may hold any data item. Any data item  $d$  that may belong to bucket  $B$  in zone  $\ell$ , may also belong to  $0\text{-child}(B)$  if  $\text{bit}_d(\ell) = 0$  or to  $1\text{-child}(B)$  if  $\text{bit}_d(\ell) = 1$ . In the following, let  $\mathcal{B}$  be the set of all buckets and let  $\text{bucket}(z, d) : \{0, \dots, \Lambda\} \times U \rightarrow \mathcal{B}$  be a function that returns the unique possible bucket of a data item  $d$  at zone  $z$ . Initially, a bucket does not contain any data. During the runtime of the system the following invariant is satisfied: Each bucket, excluding bucket  $B_\varepsilon$ , stores either 0 or between  $n$  and  $2n$  data items. Bucket  $B_\varepsilon$  stores at most  $2n$  data items.

### 2.1 Internal Storage Strategy of the Buckets

The idea of storing a set  $D$  of data items into a bucket  $B$  is to reuse the basic concepts of the storage strategy for individual layers from IRIS [7]. Roughly speaking this strategy works as follows: In order to achieve the desired robustness, we first create  $c \geq 18 \log m$  pieces  $d_1, \dots, d_c$  for each data item  $d \in D$  using Reed Solomon coding. Using  $c$  hash functions chosen uniformly and independently at random, these pieces are then mapped to servers. Finally, all these pieces are encoded with each other, such that at the end each intact server holds for each piece some parity information resulting from the encoding process. Besides encoded data pieces each bucket  $B$  additionally stores  $c$  hash functions and a timestamp  $t(B)$ . The timestamp is used to handle outdated information a server might hold if it has crashed in a previous period in which write requests were served.

In the following we roughly describe the coding strategy presented in [7]. The coding strategy is a block-based distributed strategy that follows the topology of a  $k$ -ary butterfly as described in the following. For  $k \in \mathbb{N}$  we use the notation  $[k] = \{0, \dots, k - 1\}$ .

**Definition 2.1.** *For any  $d, k \in \mathbb{N}$ , the  $d$ -dimensional  $k$ -ary butterfly  $BF(k, d)$  is a*

graph  $G = (V_k, E)$  with node set  $V_k = [d+1] \times [k]^d$  and edge set  $E$  with

$$E = \{ \{ (i, x), (i+1, (x_1, \dots, x_i, b, x_{i+2}, \dots, x_d)) \} \mid x = (x_1, \dots, x_d) \in [k]^d, i \in [d], \text{ and } b \in [k] \}.$$

A node  $u$  of the form  $(\ell, x)$  is said to be on butterfly level  $\ell$  of  $G$ . Furthermore,  $LT(u)$  is the unique  $k$ -ary tree of nodes reached from  $u$  when going downwards the butterfly (i.e., to nodes on butterfly levels  $\ell' > \ell$ ) and  $UT(u)$  is the unique  $k$ -ary tree of nodes reached from  $u$  when going upwards the butterfly. Moreover, for a node  $u$  at level  $\ell$ , let  $BF(u)$  be the unique  $k$ -ary sub-butterfly of dimension  $\ell$  ranging from butterfly level 0 to  $\ell$  in  $BF(k, d)$  that contains  $u$ .

A visualization of a  $k$ -ary butterfly is given in Figure 1.

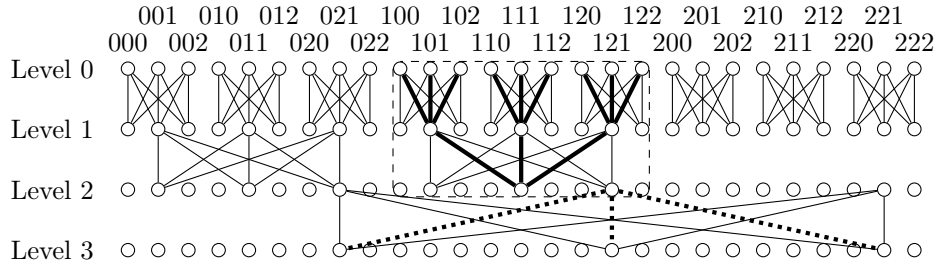


Figure 1: Visualization of a  $k$ -ary butterfly  $BF(k, d)$  for  $k = d = 3$ . For a better readability most of the edges from level two and three are omitted. The dashed box denotes the sub-butterfly  $BF((2, 111))$ . The thick solid lines in the dashed box denote the edges of  $UT((2, 111))$ . The thick dotted lines denote the edges of  $LT((2, 121))$ .

In the following let  $BF(k, d)$  be a  $k$ -ary butterfly with  $n = k^d$  and with server  $s_i$ ,  $i \in \{0, \dots, n-1\}$ , emulating the butterfly nodes  $(0, i), \dots, (d, i)$ . That is, whenever a butterfly node  $(j, i)$ ,  $j \in \{0, \dots, d\}$  is supposed to perform an action or store data, this is done by server  $s_i$ . We say a server  $s$  is *connected via the  $k$ -ary butterfly* to another server  $s'$ , if there is an edge  $(u, v)$  in the butterfly such that  $u$  is emulated by  $s$  and  $v$  is emulated by  $s'$ .

While in IRIS each server holds  $O(1)$  data pieces per layer, in our system each server holds  $O(\log n)$  data pieces per bucket. This is due to the fact that each bucket contains  $O(n)$  data items and for each data item  $c = \Theta(\log m)$  pieces are created and distributed evenly among the servers. Hence, we simply concatenate the data pieces a server  $s_i$  holds in a bucket  $B$  and denote the resulting data *block* as  $b_i$ .

In order to encode the data blocks  $b_0, \dots, b_{n-1}$  assigned to the servers  $s_0, \dots, s_{n-1}$  in bucket  $B$ , initially,  $b_i$  is placed in node  $(0, i)$  for every  $i \in \{0, \dots, n-1\}$ . Given that in butterfly level  $\ell$  we have already assigned data blocks  $d(\ell, x)$  to the nodes  $(\ell, x)$  we use the coding strategy presented in [7] to assign data blocks  $d(\ell+1, x)$  to the nodes at butterfly level  $\ell+1$ . The used coding strategy is based on some simple

parity computations and ensures the following property: If at most one butterfly node  $(\ell + 1, x_j)$  from the set of nodes  $\{(\ell + 1, x_1), \dots, (\ell + 1, x_k)\}$  is crashed, then the information in the remaining nodes  $(\ell + 1, x_i)$ ,  $i \in \{1, \dots, k\} \setminus \{j\}$ , suffices to recover  $d(\ell, x_1), \dots, d(\ell, x_k)$ . Furthermore, with Lemma 2.4 in [7] the storage amount of each server  $s_i$ ,  $i \in \{0, \dots, n - 1\}$ , required for the encoding of a single bucket is upper bounded by  $(1 + e)z$ , where  $z$  denotes the maximum size of the data blocks stored at any server  $s_j$ ,  $j \in \{0, \dots, n - 1\}$ . Since there may exist outdated data items in the system, but for each level at most one, i.e. in total at most  $\Lambda + 1 = O(\log n)$  many for each data item, the redundancy of our system increases to  $O(\log n)$ .

**Corollary 2.2.** *RoBuSt has a redundancy of  $O(\log n)$ .*

### 3 The Write Protocol

In the following let  $D$  with  $|D| \leq (1 - \delta)n$  and  $\delta < 1/72 \cdot n^{1/\log \log n}$ , be the set of data items for which intact servers received write requests. For a data item  $d$  that is stored in the system denote the  $c$  pieces that have been created from  $d$  using Reed Solomon coding as  $d_1, \dots, d_c$ . Furthermore, denote the server that is holding  $d_1$  (after the pieces have been spread over the  $n$  servers) as the server *maintaining*  $d$ .

#### 3.1 Preprocessing Stage

In this stage, for each crashed server  $s_i$ , a unique intact server is determined, denoted as the *representative* of  $s_i$ , such that at the end of this stage each crashed server is the representative of at most two other servers. The idea of the representatives is to let them take over the roles of the according crashed servers in actions (e.g. routing, computations) the crashed servers were supposed to perform. For this, we additionally need to ensure that each intact server knows the representatives of all crashed servers it is connected to in the underlying  $k$ -ary butterfly.

The determination of the representatives and the introduction of the representatives to the appropriate servers can be done in the same manner as in the *butterfly completion stage* of [7], which can be carried out in  $(2 + o(1)) \log n$  rounds with a congestion of at most  $O(\log n)$  (see Lemma 2.11 in [7]). In contrast to [7], we do not need to compute a so-called *decoding depth* here that gives information about the minimum level of the butterfly that the decoding must be initiated from, which would take  $O(\log n)$  rounds. In the following, we denote by  $s(i)$  the representative of  $s_i$  if  $s_i$  is crashed or  $s_i$  itself otherwise.

#### 3.2 Writing Stage Overview

In order to keep the specification of our system simple, we first give a high-level overview of how a set of write requests is handled. Further details are given in the following subsection.

The Writing Stage consists of up to  $\Lambda + 1$  phases. Each phase  $z \in \{0, \dots, \Lambda\}$  deals with a single bucket  $B_z$  from zone  $z$  only and receives a set of data items  $D_z$  to be



inserted into  $B_z$ . At the beginning,  $D_0 := D$  is the set of all data items for which there are write requests. In the following,  $\mathcal{D}(B_z)$  denotes the set of data items that are stored in bucket  $B_z$  (at the beginning of phase  $z$ ). Phase  $z \in \{0, \dots, \Lambda\}$  consists of the following steps.

1. Completely decode  $B_z$  and send all decoded pieces of a data item  $d \in \mathcal{D}(B_z)$  to the server maintaining  $d$  (for details, see the appendix).
2. If  $|\mathcal{D}(B_z) \cup D_z| \leq 2n$ : Add the data items from  $D_z$  to  $\mathcal{D}(B_z)$ , choose  $c$  new hash functions  $h_1, \dots, h_c : U \rightarrow V$  uniformly at random for  $B_z$ , and reencode  $B_z$  (see appendix and below).
3. Else ( $|\mathcal{D}(B_z) \cup D_z| > 2n$ ):
  - a) The intact servers agree on a subset  $D_{z+1} \subseteq \mathcal{D}(B_z) \cup D$  of size  $n$  with the property that for all  $d, d' \in D_{z+1}$ ,  $\text{bit}_d(z) = \text{bit}_{d'}(z) = b \in \{0, 1\}$  (for details, see the appendix).
  - b) Reencode the data items in  $(D_z \cup \mathcal{D}(B_z)) \setminus D_{z+1}$  in bucket  $B_z$  and choose  $c$  new hash functions  $h_1, \dots, h_c : U \rightarrow V$  uniformly at random for  $B_z$ . (see below)
  - c) Set  $B_{z+1} := 0\text{-child}(B_{z+1})$  if  $b = 0$  and  $B_{z+1} := 1\text{-child}(B_{z+1})$  if  $b = 1$  and propagate the data items in  $D_{z+1}$  to the next phase (for details, see the appendix).

Each phase of the Writing Stage can be performed in  $O(\log n)$  rounds with a congestion of  $O(\log n)$  at each server in each round (see appendix). Since there are at most  $O(\log n)$  phases in the Writing Stage, the overall runtime is  $O(\log^2 n)$  rounds.

### 3.2.1 Encoding of a Bucket

In the following we describe how a set of data items is reencoded into a bucket, as required in step 2 and step 3b. Note that the reencoding of a bucket does not only consist of the simple encoding of the data items belonging to that bucket but it consists of some additional steps, as described in the following.

First, in contrast to IRIS,  $s(1)$  chooses  $c$  hash functions  $h_1, \dots, h_c : U \rightarrow V$  uniformly at random that will be used to map data pieces of this bucket to servers. While in IRIS the hash functions that map data pieces to servers are never changed, we need to choose new hash functions for a bucket  $B$  whenever  $B$  is (re)encoded. The reason for this is that otherwise the adversary would be able to generate write requests that overload certain servers.

Note that the hash functions need to satisfy certain expansion properties, but if  $c$  is chosen sufficiently large ( $c \geq 18 \log m$ ) they do so, w.h.p. (more information is provided in the appendix). After that,  $s(1)$  distributes the  $c$  hash functions to all other intact servers  $s(i)$ . This distribution can be realized by simply broadcasting the hash functions in the  $k$ -ary butterfly from level  $\log_k n$  to level 0. In addition,  $s_1$  distributes a current timestamp  $t(B_z)$  to all other intact servers and each intact server  $s(i)$  sets its current timestamp for bucket  $B_z$  to that value. Each server  $s(i)$  now creates for each

data item which it maintains or which it has received write requests for and which are not propagated to the next phase the  $c$  pieces  $d_1, \dots, d_c$  of  $d$  using Reed Solomon coding (Section 2). Here,  $d_j$ ,  $j \in \{1, \dots, c\}$ , is supposed to be sent to the server  $s'$  responsible for  $h_j(d)$  or to its representative if  $s'$  is crashed. Unfortunately, a server  $s(i)$  does not necessarily know the representative of the server  $s'$  if that server is crashed. Thus, instead of sending the data pieces directly, the servers initiate a bottom-up routing in the underlying  $k$ -ary butterfly in order to determine the representative of  $h_j(d)$  for each  $1 \leq j \leq c$ . Obviously, this takes only  $\log_k n$  rounds and can be performed with a congestion of  $O(k)$  per node. Once  $s(i)$  knows the representative of  $h_j(d)$ , it directly sends  $d_j$  to  $h_j(d)$  for all  $1 \leq j \leq c$ .

After the pieces of data items have been distributed, the servers encode the data items in  $(\mathcal{D}(B_z) \cup D_z) \setminus D_{z+1}$  in a distributed fashion. Note that the set of data blocks for server  $i$  in zone  $z$  is completely overwritten for each server  $s(i)$  in this process. This can be done by a simple top-down approach using the coding strategy for IRIS (see Section 2.1 in [7]). In addition, we also store the timestamp of the bucket along with the data block by appending it to the composed data block.

The following lemma holds during the encoding step, regardless of the current phase.

**Lemma 3.1.** *Assume the adversary blocks less than  $(\gamma/2) \cdot 2^{\log_k n}$  servers, with  $\gamma = 1/36$ . Then, for any data item  $d$  that is (re-)written during the current period, and any level  $0 \leq \ell \leq \log_k n$ , there are at most  $c/6$  pieces of  $d$  that are mapped to sub-butterflies  $BF(v)$  (for some  $v$  at level  $\ell$ ) with at least  $\lceil 2^{\ell-1} \rceil$  crashed servers in  $BF(v)$ , w.h.p.*

*Proof.* In the following, we denote a sub-butterfly  $BF(v)$  for some  $v \in V$  at level  $\ell$  as *blocked at level  $\ell$*  if at least  $\lceil 2^{\ell-1} \rceil$  servers in  $BF(v)$  are crashed (note that we need the ceiling function only for the special case  $\ell = 0$ ). Let  $d$  be a data item, and let  $0 \leq \ell \leq \log_k n$  be a fixed level in the underlying  $k$ -ary butterfly. First of all, we show that the fraction of blocked sub-butterflies at level  $\ell$  is at most  $\gamma$ . Using the Chernoff bounds [4], we can conclude from this that the number of pieces of  $d$  that are mapped to blocked sub-butterflies are at most  $c/6$  with high probability.

Recall that each sub-butterfly at level  $\ell$  contains exactly  $k^\ell$  servers. Obviously, for  $\ell = 0$ , the fraction of crashed servers at level  $\ell$  is upper bounded by  $\gamma/2 < \gamma$ . Thus, in the following, we assume  $1 \leq \ell \leq \log_k n$ . Let  $b$  be the number of blocked sub-butterflies at level  $\ell$ . Then, there exist at least  $b \cdot 2^{\ell-1}$  crashed servers. On the other hand, the adversary can block only less than  $\gamma/2 \cdot 2^{\log_k n}$  servers. Hence,  $b \cdot 2^{\ell-1} < \gamma \cdot 2^{\log_k n - 1}$  which is equivalent to  $b < \gamma \cdot 2^{\log_k n - \ell}$ . Recall that there are exactly  $k^{\log_k n - \ell}$  sub-butterflies at level  $\ell$ . This yields that the fraction of blocked sub-butterflies at level  $\ell$  is at most  $\gamma \cdot \frac{2^{\log_k n - \ell}}{k^{\log_k n - \ell}} \leq \gamma$ . Using the Chernoff bounds it is easy to show that at most  $c/6$  pieces of  $d$  are mapped to a blocked sub-butterfly, w.h.p. □

The lemma plays an important role in the proof of the correctness of the Lookup Protocol.

## 4 The Lookup Protocol

In order to keep the specification of our system simple, we provide the description of the lookup protocol as a separate protocol that is executed after the execution of the Write Protocol. The lookup protocol is divided into two stages: the Preprocessing Stage (Section 4.1) and the Zone Examination Stage (Section 4.2). The former is similar to the Preprocessing Stage of the Write Protocol (Section 3.1). The latter is performed for each zone individually and split into two further stages: the Probing Stage and the Decoding Stage. The basic idea of the Probing Stage is to answer a request by directly collecting a sufficient number of data pieces. If this is not possible, either because too many of the servers holding a piece are crashed or because of congestion, the Decoding Stage tries to recover a data item by utilizing the distributed coding described in Section 2.1. Note that both a Probing Stage as well as a Decoding Stage can be found in IRIS ([7]), too. While they match in their general structure, there are important differences that are caused by the differences in the underlying structure and the implications of the write functionality. For example, servers may now store obsolete data items without being aware of that.

### 4.1 The Preprocessing Stage

The Preprocessing Stage is exactly the same as in Section 3.1. If at least one write request has been handled in the current period, we can thus skip this part and re-use the established  $k$ -ary butterfly and the unique representatives.

### 4.2 The Zone Examination Stage

In the following let  $\mathcal{D}$  be the set of data items for which a lookup request arrived at an intact server. The idea of this stage is to successively perform a lookup for each  $d \in \mathcal{D}$  in each zone until a copy of  $d$  has been found and returned to the appropriate server. The zone examination stage is performed for at most  $\Lambda + 1$  zones starting with zone 0.

In each phase  $z \in \{0, \dots, \Lambda\}$ , beginning with  $z = 0$ , each server with an unserved lookup request for some data item  $d$  initiates a lookup request for  $d$  in bucket  $\text{bucket}(z, d)$ . Any server that receives a copy of the data item it requested during the lookup in zone  $z$ , as described in the following, returns that copy and is finished. All remaining lookup requests are handled in the next phase, phase  $z := z + 1$ . This procedure is repeated until each lookup request is served.

Handling a set of lookup requests in one phase  $z$  is done by performing the Probing Stage and the Decoding Stage as described in the following.

#### 4.2.1 Probing Stage

In the following let  $s$  be an intact server that has an unserved lookup request for a data item  $d$  at the beginning of phase  $z$ . The idea of the Probing Stage is to either achieve  $c/3$  up-to-date pieces such that  $d$  can be recovered. Or to assign the request for  $d$  to a

level  $\{1, \dots, \log_k n\}$  (as defined later) in order to further handle the request in the next stage, the Decoding Stage. In the following, for a server  $s'$ , an index  $i \in \{1, \dots, c\}$ , and a data piece  $d'$  we denote by  $P_i(s', d')$  the unique path of length  $\log_k n$  in the  $k$ -ary butterfly from the butterfly node on level  $\log_k n$  emulated by  $s'$  to the butterfly node on level 0 emulated by the server that is responsible for  $h_i(d')$ .

On a high level view, in phase  $z$ , server  $s$  performs the following steps.

1. Acquire current hash functions and timestamp  $t_d$  for bucket  $bucket(z, d)$ .
2. Choose  $c$  intact servers  $s(d_1), \dots, s(d_c)$  uniformly and independently at random.
3. Send a **probe** $(d, i, t_d)$  message to  $s(d_i)$ ,  $i \in \{1, \dots, c\}$ , in order to initiate the forwarding of the **probe** $(\cdot)$  message along the  $c$  paths  $P_i(s(d_i), d_i)$ .

Note that acquiring the hash functions in step 1 is necessary since  $s$  may have been crashed in the last period in which a write occurred in bucket  $bucket(Z, d)$  (at which the hash functions were replaced). Acquiring the current hash functions and the timestamp works as follows: First of all,  $s$  randomly chooses  $\kappa := \Theta(\log n)$  intact servers and asks them for their timestamp in bucket  $bucket(Z, d)$ . The intact servers can be found in  $O(1)$  communication rounds, w.h.p., by selecting  $\kappa$  random servers in each round until  $\kappa$  intact servers have been found. Let  $t_d$  be the maximum timestamp  $s$  received. If  $t_d$  is greater than the timestamp  $s$  stores for  $bucket(Z, d)$ ,  $s$  knows that it does not have the current hash functions and asks one server from which it received  $t_d$  for the  $c$  hash functions for bucket  $bucket(Z, d)$ . Note that during this process each server only receives  $O(\log n)$  requests throughout this process, w.h.p.

Once  $s$  knows the correct hash functions, its goal is to retrieve at least  $c/3$  pieces of  $d$ . Since contacting the servers holding the  $c$  pieces of  $d$  directly may cause a too high congestion at these servers, we use the method of forwarding  $c$  probes from uniformly chosen intact servers  $s(d_1), \dots, s(d_c)$  to the servers responsible for the  $c$  pieces of  $d$  along the  $c$  paths  $P_1(s(d_1), d_1), \dots, P_c(s(d_c), d_c)$  (step 2, step 3). Analogously to step 1 choosing the  $c$  intact servers in step 2 takes  $O(1)$  communication rounds, w.h.p.

In the following we describe how the nodes from the paths  $P_1(s_1, d_1), \dots, P_c(s_c, d_c)$  react on incoming messages during this phase. Let  $u$  be a butterfly node on level  $\ell \in \{0, \dots, \log_k n\}$  that has received a **probe** $(d, i, t_d)$  message. In order to reduce redundancy  $u$  combines probes for the same piece of  $d$  (and thus the same target) and  $u$  marks itself as the new origin of the probe (technique of *splitting and combining* [7]). In the following we denote a butterfly node  $u$  as *congested* if it has received more than  $\alpha \cdot c$  **probe** $(\cdot)$  messages for different probes, for a sufficiently large constant  $\alpha > 0$ . Whenever  $u$  receives a **probe** $(d, i, t_d)$  message,  $u$  performs the following steps.

1. If  $u$  is congested:
  2. Stop forwarding the probe and send a **fail** $(d, i, \ell)$  message to the origin of the probe message.
3. Else:

4. If  $\ell \neq 0$ : Forward  $\text{probe}(d, i, t_d)$  message to the butterfly node on level  $\ell - 1$  on the path  $P_i(s(d_i), d_i)$ .
5. If  $\ell = 0$ : (probe has reached its destination)
6. If  $u$ 's current version of bucket  $\text{bucket}(Z, d)$  has timestamp  $t_d$  and the server emulating  $u$  is not just a representative of  $u$ :
7. If  $u$  holds piece  $d_i$  of  $d$ : Send requested piece  $d_i$  to the origin of the probe message.
8. Else: Send  $\text{notexists}(d)$  message to the origin of the probe message.
9. Else: Send  $\text{fail}(d, i, 0)$  to the origin of the probe message.

If a butterfly node on level  $\ell \in \{0, \dots, \log_k n - 1\}$  receives a data item, a  $\text{fail}(\cdot)$ , or a  $\text{notexists}(\cdot)$  message, it forwards this answer to the origin of the request to which this message was an answer to (along the same path that the request was routed). A butterfly node on level  $\log_k n$  emulated by  $s(d_i)$ ,  $i \in \{1, \dots, c\}$ , that received an answer for a probe for data piece  $d_i$  simply forwards this answer to the server that initiated the forwarding of that probe. These answers ensure that after  $O(\log_k n)$  rounds the server  $s$  that received a lookup request for a data item  $d$  has received for all initially sent  $\text{probe}(\cdot)$  messages a piece of  $d$ , or a  $\text{notexists}(d)$  message, or the level at which the probing failed. Depending on which kinds of answers  $s$  has received, it reacts as follows:

- If  $s$  received at least  $c/3$  up-to-date pieces of  $d$ ,  $s$  recovers  $d$  using Reed Solomon coding and answers the request.
- Else if  $s$  receives a  $\text{notexists}(d)$  message,  $s$  answers that the requested data item does not exist in the system.
- Else if  $s$  receives more than  $2c/3$   $\text{fail}(d, i, 0)$  messages,  $s$  declares the request for  $d$  to belong to level  $\ell$ , where  $\ell \in \{1, \dots, \log_k n\}$  is the smallest level that contains at least  $5c/6$  active probes for  $d_i$ , i.e., probes for  $d_i$  that successfully passed the probing at level  $\ell$  and all levels  $\ell' > \ell$ .

It is easy to see that the Probing Stage takes at most  $O(\log n)$  communication rounds per phase with at most  $O(\log^2 n)$  congestion at every server in each round. Note that if a data item belongs to a level  $\ell$ , then at least  $5c/6$  of its probes successfully pass level  $\ell$  and get deactivated later in the probing (i.e., in a level  $\ell' < \ell$ ). To this end, each data item can either be retrieved successfully (this is the case if  $5c/6 > c/3$  pieces pass level 0) or belongs to a level  $1 \leq \ell < \log_k n$ .

For the proof of the correctness of the protocol, the following lemma plays an important role.

**Lemma 4.1.** *If the adversary can only block less than  $(\gamma/2) \cdot 2^{\log_k n}$  servers, then for every  $\ell \in \{1, \dots, \log_k n\}$ , the number of data items belonging to level  $\ell$  is at most  $\gamma n / k^{\ell-1}$  with  $\gamma = 1/36$ .*

The general idea and structure of the proof of Lemma 4.1 is based on the proof of Lemma 2.16 in [7]. In contrast to [7], the only level at which requests are aborted due to crashed nodes is level 0. In addition, we also have to take into account that nodes may store outdated information because they were blocked at the round in which new data was written. Besides this, we have a different definition of when a node belongs to level  $\ell$  here (we require at least  $5c/6$  active probes instead of  $c/2$ ) and a different value of  $\gamma$ .

In order to prove Lemma 4.1 we need to introduce the following definitions:

**Definition 4.2** (Congested sub-butterfly). *Let  $v$  be a node at level  $\ell$  in the butterfly. The sub-butterfly  $BF(v)$  is called congested at level  $\ell$  if the servers in  $BF(v)$  receive more than  $k^\ell \alpha c/2$  probes for different  $d_i$  pieces in total when the requests are processed at level  $\ell$ .*

**Definition 4.3** (Congested data item). *A data item  $d$  is called congested at level  $\ell$  if there exist congested sub-butterflies  $BF(s_{i_1}^{(\ell_1)}(d)), \dots, BF(s_{i_r}^{(\ell_r)}(d))$  with  $\ell_i \geq \ell - 1$ ,  $r = c/6$ , and  $i_1, \dots, i_r$  being pairwise different.*

As a crucial ingredient for the proof of Lemma 4.1, we require the hash functions  $h_1, \dots, h_c$  to satisfy a certain expansion property, which holds if the hash functions are chosen uniformly and independently at random, w.h.p.. For this, we need the following definitions.

**Definition 4.4** ( $b$ -bundle). *Given a set  $S \subset U$  of keys and a  $k \in \mathbb{N}$ , we call  $F \subseteq S \times \{1, \dots, c\}$  a  $b$ -bundle of  $S$  if every  $d \in S$  has exactly  $b$  many pairs  $(d, i)$  in  $F$ .*

**Definition 4.5**  $((b, \sigma)$ -expander). *For any sub-butterfly  $B$  let  $V(B)$  be the set of servers emulating the nodes of  $B$ . Let  $\mathcal{H}$  be a collection of hash functions  $h_1, \dots, h_c$ . Given  $h_1, \dots, h_c$  and a level  $\ell \in \{0, \dots, \log_k n\}$ , we define  $\Gamma_{F, \ell}(S) := \bigcup_{(d, i) \in F} V(BF(s_i^{(\ell)}(d)))$ . Given a  $0 < \sigma < 1$ , we call  $\mathcal{H}$  a  $(b, \sigma)$ -expander if for any  $0 \leq \ell < \log_k n$ , any  $S \subseteq U$  with  $|S| \leq \sigma n/k^\ell$ , and any  $b$ -bundle  $F$  of  $S$ , it holds that  $|\Gamma_{F, \ell}(S)| \geq k^\ell |S|$ .*

The following Claim can be proven analogously to Claim 2.13 of [2].

**Claim 4.6.** *If the hash functions  $\mathcal{H} = \{h_1, \dots, h_c\}$  are chosen uniformly and independently at random,  $m = |U|$  sufficiently large, and  $c \geq 18 \log m$ , then  $\mathcal{H}$  is a  $(c/6, 1/36)$ -expander, w.h.p.*

**Proof of Lemma 4.1:** For the proof, we distinguish between level 1 and all other levels  $\ell > 1$ . To simplify, whenever we say that a piece  $d_i$  of a data item  $d$  is *aborted at level  $\ell$* , we mean that the probing for  $d_i$  did not successfully pass level  $\ell$  (but was answered with a  $\text{fail}(d, i, \ell)$  message). Recall that there are two reasons for a request for a piece  $d_i$  to be aborted: Either due to an excessive congestion (at any node at level  $\ell > 0$ ) or because the server responsible for  $d_i$  is crashed or outdated (at level 0).

Note that whenever a data item  $d$  belongs to level 1, then more than  $3c/6$  of the  $c$  pieces of  $d$  must have been aborted at level 0 due to outdated or crashed nodes at

level 0 ( $5c/6$  pieces of  $d$  successfully passed level 1 by the definition of when a data item belongs to level 1 and if at least  $c/3$  pieces would have passed level 0 successfully, the data item would have been answered already). First of all, Lemma 3.1 yields that at most  $c/6$  of the  $c$  pieces of any data item can have been aborted at level 0 due to outdated nodes. Thus, for any data item  $d$  belonging to level 1, more than  $c/6$  pieces of  $d$  must be aborted at level 0 due to crashed nodes. We will now bound the maximum number of these data items. Let  $S$  be a maximum set of data items that belong to level 1. We will show:  $|S| < \gamma n$ . We now construct a set  $F$  in the following way: for each  $d \in S$ , we choose  $c/6$  indices  $i$  with the property that  $d_i$  is aborted at level 0 due to crashed nodes and add these  $(d, i)$  to  $F$ . Note that  $F$  is a  $c/6$ -bundle  $F$  of  $S$ . Since the adversary can block only less than  $\gamma n$  servers, the number of servers covered by all  $BF(s_i^{(0)}(d))$  with  $(d, i) \in F$  is less than  $\gamma n$ . Since  $\Gamma_{F,0}(S)$  is exactly the set of these servers, it holds:  $|\Gamma_{F,0}(S)| < \gamma n$ . On the other hand, we know from Claim 4.6 that for any  $c/6$ -bundle  $F'$  of  $S'$  with  $|S'| \leq \gamma n$ ,  $|\Gamma_{F',0}(S')| \geq |S'|$ . Note that this also implies that for any  $c/6$ -bundle  $F'$  of  $S'$  with  $|S'| \geq \gamma n$ ,  $|\Gamma_{F',0}(S')| \geq \gamma n$ . Now, assume for contradiction that  $|S| \geq \gamma n$ . This yields  $|\Gamma_{F',0}(S')| \geq \gamma n$ . Since this is a contradiction to what we said before,  $|S| < \gamma n$  must hold and thus the number of data items belonging to level 1 must be bounded by  $\gamma n$ .

Next, for any level  $\ell > 1$ , we bound the number of data items belonging to level  $\ell$ . First of all, note that the only reason for a piece of data item to be aborted on a level  $\ell \geq 1$  is due to congestion at a node at level  $\ell$ . Second, note that it can be shown that whenever a  $\text{probe}(d, i, t_d)$  is aborted on level  $\ell \geq 1$  due to congestion, then  $BF(s_i^{(\ell)}(d))$  is congested w.h.p. (see Claim 2.18 of [7]). Thus, whenever a data item  $d$  is declared to belong to a level  $\ell > 1$ , then at least  $c/6$   $\text{probe}(d, i, t_d)$  messages have been deactivated at level  $\ell - 1$  or higher because of congested sub-butterflies, i.e.  $d$  is congested at level  $\ell - 1$  (see Def. 4.3). Thus, if many data items belong to level  $\ell$ , then many sub-butterflies must be congested at level  $\ell - 1$ . However, as we will prove, only a constant fraction of the sub-butterflies can be congested at level  $\ell - 1$ , which implies that only a constant fraction of all data items can belong to level  $\ell$ .

Fix  $1 < \ell \leq \log_k n$ . As mentioned before, we will now bound the number of data items that are congested at level  $\ell - 1$ . Let  $S$  be a maximum set of data items that are congested at level  $\ell - 1$ . We will show:  $|S| < \gamma n / k^{\ell-1}$ . Again, we construct a  $c/6$ -bundle  $F$  of  $S$  (adding, for each  $d \in S$ ,  $c/6$  indices  $i$  to  $F$  with the property that  $BF(s_i^{(l_i)}(d))$  is congested). We first show that for  $\alpha$  sufficiently large, less than a fraction of  $\gamma$  of all butterflies at level  $\ell - 1$  can be congested. Recall that a sub-butterfly on level  $\ell - 1$  is congested if it receives more than  $\alpha c k^{\ell-1} / 2$  probes for different  $(d, i)$ -pairs. Let  $\delta$  be the maximum fraction of servers the adversary may block. Since there are at most  $(1 - \delta)n$  lookup requests in total, at most  $c(1 - \delta)n$  probes arrive at level  $\ell - 1$ . Thus, at most  $c(1 - \delta)n / (\alpha c k^{\ell-1} / 2) = 2(1 - \delta)n / (\alpha k^{\ell-1})$  sub-butterflies can be congested at level  $\ell - 1$ . Since there are exactly  $n / k^{\ell-1}$  disjoint sub-butterflies at level  $\ell - 1$ , the fraction of congested sub-butterflies at level  $\ell - 1$  is upper bounded by  $2(1 - \delta) / \alpha \leq 2 / \alpha$ . Hence, for  $\alpha > 2 / \gamma$ , less than a  $\gamma$ -fraction of the sub-butterflies on level  $\ell - 1$  can be congested. That is, all of the congested sub-butterflies  $BF(s_i^{(l_i)}(d))$  with  $(d, i) \in F$  together contain less than a  $\gamma$ -fraction of the sub-butterflies on level

$\ell - 1$ . This implies  $|\Gamma_{F,\ell-1}(S)| < \gamma n$ .

On the other hand, from Claim 4.6, we can deduce that for any  $c/6$ -bundle  $F$  of  $S'$  with  $|S'| \geq \gamma n/k^{\ell-1}$ ,  $|\Gamma_{F,\ell-1}(S')| \geq \gamma n$ . By assuming for contradiction that  $|S| \geq \gamma n/k^{\ell-1}$ , we can deduce that  $|\Gamma_{F,\ell-1}(S)| \geq \gamma n$ , which is a contradiction in this case, too. Thus,  $|S| < \gamma n/k^{\ell-1}$ .

Therefore, less than  $\gamma n/k^{\ell-1}$  are congested at level  $\ell - 1$ . For the remaining data items, at least  $5c/6$  pieces are not congested. Thus, these data items do not belong to level  $\ell$ . This finishes the proof.  $\square$

#### 4.2.2 Decoding Stage

The Decoding stage proceeds in  $\log_k n$  sub-phases. In the following, for a server  $s$  that holds a lookup request for some data item  $d$  that has not been answered before this sub-phase, we define  $s_i^{(\ell)}(d)$  as the node at level  $\ell$  on the unique path of length  $\log_k n$  from the butterfly node on level  $\log_k n$  emulated by  $s_i(d)$  to the butterfly node on level 0 responsible for  $h_i(d)$ .

On a high level view, the Decoding Stage works as follows: During each sub-phase  $1 \leq \ell \leq \log_k n$ , starting with level 1, we try to recover the data items belonging to level  $\ell$ . In order to recover a data item  $d$ , we need to collect at least  $c/3$  pieces of  $d$ . To do so, we randomly choose  $5c/6$  requests for pieces of  $d$  that were active at level  $\ell$  in the Probing Stage and for each of these pieces  $d_i$  we determine whether  $BF(s_i^{(\ell)}(d))$  can be decoded without congestion (as described later). If  $BF(s_i^{(\ell)}(d))$  can be decoded without congestion, the decoding is initiated and the result of this is sent back to the origin. (Throughout the whole process, we use the same combining/splitting approach of messages as in the Probing Stage.) Otherwise, the origin is informed that the according piece of  $d$  could not be decoded. If for a data item  $d$  not sufficiently many (i.e., less than  $c/3$ ) pieces could be recovered, the request for  $d$  is declared to belong to level  $\ell + 1$  and will be considered again in the next sub-phase. Note that requests for non-existing data items may be handled in the Decoding Stage. However, these can be treated as existing items (with the only difference being that one intact server taking part in the decoding is sufficient to tell that the data item does not exist).

In the following, we describe the operation of any sub-phase  $\ell$  in more detail. First of all, each server  $s$  that is responsible for a lookup request of a data item  $d$  that belongs to level  $\ell$  chooses  $5c/6$  among the at least  $5c/6$  indices of pieces of  $d$  that were active at level  $\ell$  in the Probing Stage. For such a piece  $d_i$  of  $d$  with current timestamp  $t$ ,  $s$  sends a `decode`( $d, i, t$ ) message from  $s_i^{(\log_k n)}(d)$  to  $v := s_i^{(\ell)}(d)$  (which is done by simply routing through the  $k$ -ary butterfly into the direction of  $h_i(d)$  for  $\ell$  rounds). In order to determine whether  $BF(v)$  can be decoded without congestion,  $v$  first checks whether it is congested, i.e., it received more than  $\beta ck$  `decode`( $\cdot$ ) messages for a sufficiently large constant  $\beta$  and, if not, then issues a `decodeCHECK`( $d, i$ ) message, which is spread to all nodes in  $UT(v)$ . During this spreading, whenever a further forwarding of all messages received by a node  $u$  at a level  $\ell - \kappa$ ,  $1 \leq \kappa < \ell$ , could lead to congestion (i.e.,  $u$  received more than  $\beta ck$  `decodeCHECK`( $d', i'$ ) messages for distinct  $(d', i')$  pairs),  $u$  stops the forwarding of all messages and instead spreads a `cong`( $\cdot$ ) message in  $BF(u)$ .



In addition, it sends a  $\text{fail}(\cdot)$  message to all neighbors at level  $\ell - \kappa + 1$ . Each node on a level  $\ell'$ ,  $\ell - \kappa + 1 \leq \ell' < \ell$ , that receives such a  $\text{fail}(\cdot)$  message forwards this message to all neighbors at level  $\ell' + 1$  from which it received a  $\text{decodeCHECK}(\cdot)$  message. By this it is ensured that whenever a node in  $BF(u)$  is congested each node  $v'$  at level  $\ell$  with  $v' \in BF(u)$  receives a  $\text{fail}(\cdot)$  message after at most  $2\ell$  rounds. Each node  $u'$  at level  $\ell - \kappa$ ,  $1 \leq \kappa < \ell$ , that received a  $\text{cong}()$  message initiates the same spreading of  $\text{cong}()$  messages in  $UT(u')$ . If  $v$  had not been congested before the spreading and  $v$  has not received any  $\text{fail}(\cdot)$  message after  $2\ell$  rounds, it knows that any piece of a data item for which  $v$  received a  $\text{decode}(\cdot)$  message can be decoded if not outdated nodes in  $BF(v)$  forbid this. Thus, it initiates the decoding for each of the pieces, which may fail due to outdated nodes. If the decoding is possible, it recovers all of these pieces within  $O(\ell)$  communication rounds with a congestion of at most  $\beta ck^2$  per node (using the distributed decoding described in [7]). These are then forwarded to the origins of the requests. If, however the decoding fails, or if  $v$  was congested or received a  $\text{fail}(\cdot)$  message, it sends a  $\text{fail}(\cdot)$  message to the origins of the  $\text{decode}(\cdot)$  messages it received (which, again, are forwarded up to the initiator of that  $\text{decode}(\cdot)$  message). Finally, if a server  $s$  that is responsible for a lookup request of a data item  $d$  receives at least  $c/3$  successfully decoded pieces, it determines  $d$  and answers the request. Otherwise, it changes the request to belong to level  $\ell + 1$  such that it will be processed again in the next sub-phase.

It is easy to see that the Decoding Stage satisfies the following property:

**Lemma 4.7.** *The Decoding Stage takes at most  $O(\log n)$  communication rounds per sub-phase with at most  $O(\log^3 n)$  congestion in every node at each round, w.h.p.*

Similarly to Lemma 4.1 of the Probing Stage, for the Decoding Stage the following lemma holds:

**Lemma 4.8.** *At the beginning of each sub-phase  $\ell \in \{1, \dots, \log_k n\}$ , the number of data items with requests belonging to level  $\ell$  is at most  $\varphi n/k^\ell$  with  $\varphi = \Theta(k)$ .*

For the proof of Lemma 4.8 we need the following definitions:

**Definition 4.9** (Blocked sub-butterfly). *Let  $v$  be a node at level  $\ell$  in the butterfly. The sub-butterfly  $BF(v)$  is called blocked at level  $\ell$  if at least  $2^{\ell-1}$  servers from  $BF(v)$  are crashed.*

**Definition 4.10** (Congested sub-butterfly). *Let  $v$  be a node at level  $\ell$  in the butterfly. The sub-butterfly  $BF(v)$  is called congested at level  $\ell$  if the servers in  $BF(v)$  receive more than  $\beta ck$  requests for different  $d_i$  pieces in total when the requests are processed at level  $\ell$ .*

**Definition 4.11** (Blocked/Congested data item). *A data item  $d$  is called blocked/congested at level  $\ell$  if there exist blocked/congested sub-butterflies  $BF(s_{i_1}^{(\ell_1)}(d)), \dots, BF(s_{i_r}^{(\ell_r)}(d))$  with  $\ell_i \geq \ell$ ,  $r = c/6$ , and  $i_1, \dots, i_r$  being pairwise different.*

Furthermore, we need the following claims.

**Claim 4.12.** *For any data item  $d$  which is neither blocked nor congested, at most  $c/6$  pieces of  $d$  can fail due to outdated servers.*

*Proof.* Assume a data item  $d$  is neither congested nor blocked. This means that less than  $c/6$  pieces of  $d$  are congested and less than  $c/6$  pieces of  $d$  are blocked. The latter implies that the adversary blocks less than  $2^{\ell-1}$  servers from the sub-BFs  $BF(s_{i_1}^{(l_1)}(d)), BF(s_{i_2}^{(l_1)}(d)), \dots$  for the remaining pieces  $d_{i_1}, d_{i_2}, \dots$  of  $d$ . By Lemma 3.1, for all but  $c/6$  of these pieces, less than  $2^{\ell-1}$  of the servers in  $BF(s_{i_1}^{(l_1)}(d)), BF(s_{i_2}^{(l_1)}(d)), \dots$  can be outdated regarding  $d$ . Thus, for all but  $c/6$  of these pieces, less than  $2^{\ell-1} + 2^{\ell-1} = 2^\ell$  servers can be crashed or outdated, which, by Claim 2.17 of [7] means that these pieces can be recovered. Thus, at most  $c/6$  pieces of the data items that are neither blocked nor congested can fail due to outdated servers.  $\square$

Now we are ready to prove Lemma 4.8. The proof is similar to the proof of Lemma 2.21 of [7] with the main difference being that we additionally need to handle outdated data items here.

**Proof of Lemma 4.8:** In the following, let  $\gamma = 1/36$  and  $\varphi = 3\gamma k$ . We prove the lemma by induction on  $\ell$ . The basis ( $\ell = 1$ ) holds by Lemma 4.1. For the induction step, let  $\ell \in \{1, \dots, \log_k n - 1\}$  and assume that the induction hypothesis holds for level  $\ell$ . We show that the number of data items that will be propagated to level  $\ell + 1$  during sub-phase  $\ell$  is at most  $2\gamma n/k^\ell$ . Together with Lemma 4.1, this means that at the beginning of sub-phase  $\ell + 1$ , at most  $\gamma n/k^\ell + 2\gamma n/k^\ell$  data items belong to level  $\ell + 1$ , which is equal to  $\varphi n/k^{\ell+1}$  and thus proves the induction step.

Recall that in sub-phase  $\ell$ , requests for  $5c/6$  pieces of each data item belonging to level  $\ell$  are sent. Note that any request for a piece  $d_i$  of a data item  $d$  in sub-phase  $\ell$  of the Decoding Stage can only be aborted for one of the following three reasons: First, that too many servers storing information about  $d_i$  are crashed in the current period. Second, that too many servers storing information about  $d_i$  are outdated (i.e., they were crashed when the bucket storing  $d$  was last updated). Third, due to congestion in sub-phase  $\ell$  of the Decoding Stage. However, it can be shown that if at least  $c/6$  requests for a data item  $d$  are aborted during the decoding in sub-phase  $\ell$  due to too many crashed nodes, then  $d$  is blocked at level  $\ell$  and if at least  $c/6$  requests for a data item  $d$  are aborted during the decoding in sub-phase  $\ell$  due to congestion, then  $d$  is congested at level  $\ell$  w.h.p. The former claim is an implication of Claim 2.17 of [7], and the latter follows by definition and the algorithm performed in the decoding stage. Claim 4.12 now implies that for the data items which are neither blocked nor congested, at least  $5c/6 - c/6 - c/6 - c/6 = c/3$  pieces can be recovered correctly, which means that they can be answered after sub-phase  $\ell$ . Thus, in the following, we will show that at most  $\gamma n/k^\ell$  data items are blocked at level  $\ell$  and that at most  $\gamma n/k^\ell$  data items are congested at level  $\ell$ .

First of all, we prove that the number of blocked data items in sub-phase  $\ell$  is upper bounded by  $\gamma n/k^\ell$ . Let  $S$  be a maximum set of data items that are blocked at level  $\ell$ . We will show:  $|S| < \gamma n/k^\ell$ . Recall that a data item  $d$  is blocked at level  $\ell$  if there exist at least  $r = c/6$  sub-butterflies  $BF(s_{i_1}^{(\ell_1)}(d)), \dots, BF(s_{i_r}^{(\ell_r)}(d))$  with  $\ell_i \geq \ell$ , and

$i_1, \dots, i_r$  being pairwise different that are blocked, i.e., each of them contains at least  $2^{\ell_i-1}$  crashed servers. For each  $d \in S$ , let  $d_{i_1}, \dots, d_{i_r}$  be  $c/6$  such indices fulfilling this property. Further, let  $(d, d_{i_1}), \dots, (d, d_{i_r}) \in F$  for all  $d \in S$ . Then,  $F$  is a  $c/6$ -bundle of  $S$ . Since a sub-butterfly of level  $\ell'$  contains  $k^{\ell'}$  servers in total, and since a blocked sub-butterfly of level  $\ell'$  contains at least  $2^{\ell'-1}$  crashed nodes, a  $2^{\ell'-1}/k^{\ell'}$  fraction of the servers of a blocked sub-butterfly of level  $\ell'$  are crashed, which is at least  $2^{\log_k n-1}/n$  for any  $1 \leq \ell' \leq \log_k n$ . Therefore, if the adversary can only block less than  $(\gamma/2) \cdot 2^{\log_k n}$  servers, then the number of servers covered by all  $BF(s_i^{(\ell_i)}(d))$  with  $(d, i) \in F$  must be less than  $\gamma n$ . Since  $\Gamma_{F,\ell}(S)$  is exactly the set of these servers, it holds:  $|\Gamma_{F,\ell}(S)| < \gamma n$ .

On the other hand, we know from Claim 4.6 that for any  $c/6$ -bundle  $F'$  of  $S'$  with  $|S'| \leq (1/36)n/k^\ell$ ,  $|\Gamma_{F',\ell}(S')| \geq |S'|k^\ell$ . Since  $\gamma = 1/36$ , this implies that for any  $c/6$ -bundle  $F'$  of  $S'$  with  $|S'| \geq \gamma n/k^\ell$ ,  $|\Gamma_{F',\ell}(S')| \geq \gamma n$ . Now, assume for contradiction that  $|S| \geq \gamma n/k^\ell$ . This yields  $|\Gamma_{F,\ell}(S)| \geq \gamma n$ , which is a contradiction to what we said before. Hence, the number of blocked data items at level  $\ell$  is less than  $\gamma n/k^\ell$ .

For the upper bound on the number of congested data items, recall that we denote a sub-butterfly  $BF(v)$  of a node  $v$  as congested if the servers in  $BF(v)$  receive more than  $\beta ck$  decode messages for different  $(d, i)$ -pairs. For  $\beta := 3$ , it holds that  $\beta ck > 5\varphi c/(6\gamma)$ , which implies that a congested sub-butterfly  $BF(v)$  of a node  $v$  receives more than  $5\varphi c/(6\gamma)$   $\text{decode}(d, i, t)$  messages for different  $d$  and  $i$ . By the induction hypothesis and due to the fact that we send  $5c/6 \text{ decode}(\cdot)$  messages per data item, there are at most  $5c/6 \cdot \varphi n/k^\ell \text{ decode}(\cdot)$  messages in total, which means that there are less than  $\varphi n/k^\ell \cdot 5c/6 \cdot 6\gamma/(5c\varphi) = \gamma n/k^\ell$  congested sub-butterflies of dimension  $\ell$ .

Let  $S$  be a set of data items congested at level  $\ell$ . Similar to the previous part about blocked data items, we can construct a  $c/6$ -bundle  $F$  for  $S$ . Since there are less than  $\gamma n/k^\ell$  congested sub-butterflies of dimension  $\ell$  and since each sub-butterfly of dimension  $\ell$  contains  $k^\ell$  nodes,  $|\Gamma_{F,\ell}(S)| < \gamma n$ . On the other hand, if we assume  $|S| \geq \gamma n/k^\ell$ , Claim 4.6 yields  $|\Gamma_{F,\ell}(S)| \geq \gamma n$ . Since this is a contradiction, we have that the number of data items congested at level  $\ell$  is less than  $\gamma n/k^\ell$ .

As stated at the beginning of the proof, this is sufficient to prove the induction step and thus completes the proof of the lemma.  $\square$

The previous lemmas and results imply Corollary 4.13, which proves Theorem 1.1.

**Corollary 4.13.** *RoBuSt correctly serves any set of lookup and write requests (with one request per intact server) in at most  $O(\log^4 n)$  communications rounds, with a congestion of at most  $O(\log^3 n)$  at every server in each round and a redundancy of  $O(\log n)$  if less than  $1/72 \cdot n^{1/\log \log n}$  servers are crashed.*

## 5 Conclusion and Future Work

We presented the first scalable distributed storage system that is provably robust against batch-based crash failures with up to  $\gamma n^{1/\log \log n}$  crashes allowed ( $\gamma > 0$  constant). An interesting question that has not been investigated in this work is

whether the techniques that enabled the Enhanced IRIS system [7] to tolerate a larger number of failed servers could be adapted for RoBuSt in order to increase the number of crashed servers allowed up to  $\varrho n$  (for some constant  $\varrho > 0$ ) while (as a minor drawback) also increasing the redundancy to  $O(\log n)$ , such as it is the case in Enhanced IRIS.

Moreover, while we assume batch-based failures, it would be interesting to see whether a scalable distributed storage system can be designed that can tolerate failures occurring at arbitrary points in time. Dealing with a similar issue, it would also be interesting to enhance our system to allow dynamics (i.e. joins and leaves of servers) in our system in order to model P2P networks.

A further interesting challenge is to enhance our distributed storage system such that additional types of attacks can be handled, for example Byzantine attacks.

## References

- [1] Baruch Awerbuch and Christian Scheideler. A Denial-of-Service Resistant DHT. In *Proc. of DISC*, pages 33–47, 2007.
- [2] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A dos-resilient information system for dynamic data management. In *Proc. SPAA*, pages 300–309, 2009.
- [3] Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark Thober. Pagoda: A Dynamic Overlay Network for Routing, Data Management, and Multicasting. In *Proc. of SPAA*, pages 170–179, 2004.
- [4] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sums of observations. *Annals of Mathematical Statistics*, 23:409–507, 1952.
- [5] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, February 1991.
- [6] P. Druschel and A. Rowstron. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware*, pages 329–350, 2001.
- [7] Martina Eikel and Christian Scheideler. IRIS: A Robust Information System Against Insider DoS-Attacks. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA’13, pages 119–129. ACM, 2013.
- [8] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC ’01, pages 170–179, New York, NY, USA, 2001. ACM.
- [9] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USITS*, page 9, 2003.

- [10] F. Kargl, J. Maier, and M. Weber. Protecting Web Servers from Distributed Denial of Service Attacks. In *Proc. of WWW*, pages 514–524, 2001.
- [11] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. of SIGCOMM*, pages 61–72, 2002.
- [12] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.
- [13] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein. Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers. In *Proc. of CCS*, pages 8–19, 2003.
- [14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of SIGCOMM*, pages 161–172, 2001.
- [15] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 223–238, New York, NY, USA, 1989. ACM.
- [16] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Kalakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Technical Report MIT*, 2002.

## Appendix

### 5.1 Details on Decoding

At the beginning of each phase of the Writing Stage, for each data item  $d$  belonging to the current bucket  $B_z$ , all pieces of  $d$  are decoded and sent to the server maintaining  $d$ . This can be done by a bottom-up approach that proceeds in  $\log_k n + 1$  rounds: First of all, note that for each bucket, each server stores the timestamp of when it last took part in a Writing Stage for this bucket. These timestamps enable a server to identify that it stores outdated information about a bucket.

In round  $r \in \{0, \dots, \log_k n - 1\}$ , each node  $u$  at level  $\log_k n - r$  in the  $k$ -ary butterfly forwards all decoding information about a node  $v$  at level  $\log_k n - r - 1$  to  $v$ . In any round  $r' \in \{1, \dots, \log_k n\}$  any crashed node  $v$  at level  $\log_k n - r'$  filters out all messages whose timestamp is not a highest among those received and, if the number of remaining messages is at least  $k - 1$ , it can use these messages to decode enough information to function as an intact node from now on. Note that if  $v$  received a timestamp higher than its own one, it behaves like a crashed one from now on.

Any node  $w$  at level 0 that is still crashed after round  $\log_k n - 1$  can restore the pieces of data items in  $w$  by the messages it receives from the nodes at level 1. This is due to the following: First of all, the adversary can block less than  $\frac{1}{2} \cdot 2^{\log_k n}$  servers

in the current period only. Secondly, less than  $\frac{1}{2} \cdot 2^{\log_k n}$  servers can store outdated information about bucket  $B_z$ , for the same reason. Thus, less than  $2^{\log_k n}$  servers can have none or outdated pieces of data items. Then, from Claim 2.17 of [7] it follows that all pieces can be recovered at level 0.

Note that the  $c$  hash functions for the pieces of a data items were been chosen uniformly and independently at random when  $B_z$  was encoded, and after the adversary had decided on the set of blocked servers. Thus, each server holds  $O(\log n)$  pieces for bucket  $B_z$ , w.h.p. Furthermore, each server maintains at most one data item, w.h.p. This implies that the above process yields a congestion of at most  $O(\log n)$  w.h.p.

All in all, we have:

**Lemma 5.1.** *After  $\log_k n + 1$  rounds with a congestion of  $O(\log n)$  at each server in each round w.h.p., each server  $s(i)$  maintaining a data item  $d$  in  $B_z$  completely knows  $d$ , w.h.p.*

## 5.2 Details on Counting and Selection

In the following we describe the process of determining the number of data items in  $\mathcal{D}(B_z) \cup D_z$  and the elements of the set  $D_{z+1}$  (if necessary) for a phase  $z$  in a distributed fashion in more detail. For the set  $D_z$  of data items to be inserted into  $B_z$ , we denote by  $D_{z,i} \subseteq D_z$  the set of data items with a write request at server  $s(i)$ . Furthermore, we denote by  $B_{z,i} \subseteq \mathcal{D}(B_z)$  the set of data items from bucket  $B_z$  that server  $s(i)$  maintains.

First of all, each server  $s(i)$ ,  $i \in \{1, \dots, n\}$  initializes a tuple  $(\text{num}_0, \text{num}_1)$  where  $\text{num}_j$ ,  $j \in \{0, 1\}$ , is the number of data items  $d \in B_{z,i} \cup D_{z,i}$  with  $\text{bit}_d(z+1) = j$ . These tuples are now forwarded bottom up in the underlying  $k$ -ary butterfly, where each intermediate node sums up all tuples it received and forwards the result to the next smaller level. More precisely, each server  $s(i)$  first sends its tuple  $(\text{num}_0, \text{num}_1)$  to each of the  $k$  neighbors of the node  $(\log_k n, i)$  in the underlying  $k$ -ary butterfly. Any intermediate node  $v$  on level  $\ell$ ,  $0 < \ell < \log_k n$ , sets  $\text{num}_j$ ,  $j \in \{0, 1\}$ , as the sum of all  $k$  received  $\text{num}_j$ -values and sends the tuple  $(\text{num}_0, \text{num}_1)$  to its  $k$  neighbors on level  $\ell - 1$  in the underlying  $k$ -ary butterfly. Finally, a server  $s(i)$  on level 0, sums up all tuples received from neighbors on level 1 and stores the result in  $(\text{num}'_0, \text{num}'_1)$ . The following lemma is easy to check.

**Lemma 5.2.** *After  $\log_k n$  rounds, each server  $s(i)$  knows the number of data items  $d \in \mathcal{D}(B_z) \cup D_z$  with  $\text{bit}_d(z+1) = j$  for all  $j \in \{0, 1\}$ . Additionally, in every round, each server  $s$  sends and receives at most  $2k$  messages.*

The servers can now compute  $\text{size}(B_z) := \text{num}'_0 + \text{num}'_1$  and check whether  $\text{size}(B_z) > 2n$ . If this is not the case, the current bucket is reencoded together with the items from  $D_r$  (see Section 3.2.1) and the Writing Stage finished. Otherwise the servers need to degree on the bucket and a set  $D_{z+1}$  of  $n$  data items from  $\mathcal{D}(B_z) \cup D_z$  to be handled in the next phase. Whether this bucket is either  $0\text{-child}(B_z)$  or  $1\text{-child}(B_z)$  depends on the number of data items with the same  $(z+1)$ -st bit in  $D_z \cup \mathcal{D}(B_z)$ . I.e., if  $\text{num}'_0 > n$  the next bucket is  $B_{z+1} := 0\text{-child}(B_z)$  and we set  $j := 0$ , otherwise the next bucket

is  $B_{z+1} := 1\text{-child}(B_z)$  and we set  $j := 1$ . Then, since  $\text{size}(B_z) > 2n$ ,  $\text{num}'_j > n$  must hold.

In the following, the servers determine the set  $D_{z+1}$  of  $n$  data items that will be propagated to bucket  $B_{z+1}$ , as required in step 3c. This is done by a top-down approach in the tree  $LT((0,0))$  of the  $k$ -ary butterfly. In the following, we assume that each node  $v$  in  $LT((0,0))$  during the first part (the bottom-up counting) stored the tuples  $(t_{1,0}, t_{1,1}), (t_{2,0}, t_{2,1}), \dots, (t_{k,0}, t_{k,1})$  it received from its children  $v_1, \dots, v_k$  in  $LT((0,0))$  and is now still able to determine the value of  $t_{i,j}$ ,  $i \in \{1, \dots, k\}$ . Furthermore, the nodes exchange two different types of messages during this step: **full** and **partly(x)**,  $x \in \mathbb{N}$ . At the beginning,  $(0,0)$  issues **partly(size( $B_z$ ))** on itself. Depending on the message a node  $v$  receives, it performs the actions described in the following.

**partly(x):** If  $v$  is not on level  $\log_k n$ , let  $v_1, \dots, v_k$  denote the children of  $v$  in  $LT((0,0))$ . Determine the greatest index  $b$  such that  $y \leq x$  with  $y := \sum_{i=1}^b t_{i,j}$ . Send **full** to  $v_1, \dots, v_b$ . If  $x - y > 0$ , send **partly(x-y)** to  $v_{b+1}$ . If  $v$  is on level  $\log_k n$ , the server emulating  $v$  randomly chooses  $x$  data items  $d \in B_{z,i} \cup D_{z,i}$  with  $\text{bit}_d(z+1) = j$ . These data items belong to  $D_{z+1}$  and will be handled in the next phase as if the server emulating  $v$  has a new write request for them.

**full:** If  $v$  is not on level  $\log_k n$ ,  $v$  sends a **full**-message to each of its children in  $LT((0,0))$ . If  $v$  is on level  $\log_k n$ , the server emulating  $v$  removes all data items  $d \in B_{z,i} \cup D_{z,i}$  with  $\text{bit}_d(z+1) = j$ . These data items belong to  $D_{z+1}$  and will be handled in the next phase as if the server emulating  $v$  has a new write request for them.

The following lemma is easy to check:

**Lemma 5.3.** *After  $\log_k n$  additional rounds it holds:*

1. *Each server  $s(i)$  knows which of the data items in  $B_{z,i} \cup D_{z,i}$  are supposed to be encoded in bucket  $B_z$  again and which of them are propagated to the next phase.*
2. *In every round, each server  $s$  sends and receives at most  $2k$  messages.*
3. *The number of data items that are decided to belong to bucket  $B_z$  (and thus will be encoded in this bucket) is at most  $2n$ .*

It remains to distribute for each data item from  $D_{z+1}$  a write request among the  $n$  servers such that each server  $s(i)$  is responsible for exactly one of these write requests. This distribution can easily be achieved by using standard techniques for load balancing in the butterfly in  $O(\log n)$  rounds and a congestion of  $O(\log n)$  at each server.

### 5.3 Figures & Glossary

Figure 2 and Figure 3 visualize the Probing and Decoding Stage.

Table 1 provides an overview of the variables and terms used in this work and their meanings.

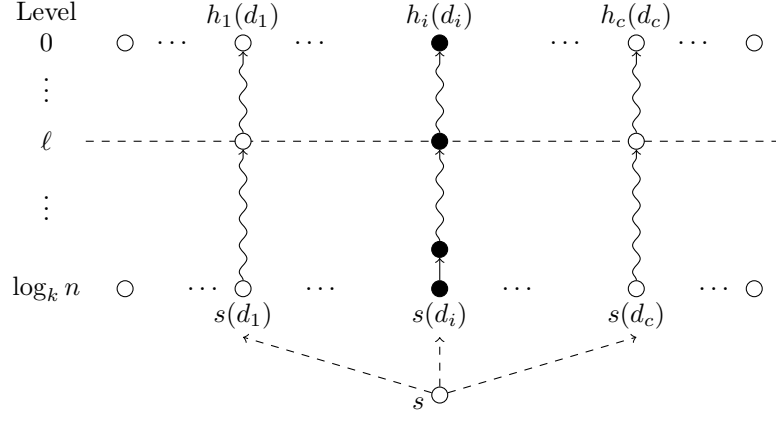


Figure 2: Visualization of the Probing Stage. The curved paths denote the paths  $P_i(s(d_i), d_i)$ .

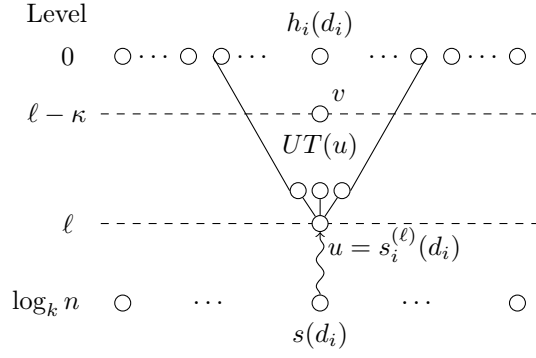


Figure 3: Visualization of sub-phase  $\ell$  of the decoding stage



Variables and Terms	Meaning and Notes
$n$	number of servers in the system
$m$	size of universe of all possible keys, polynomial in $n$
$c$	set to value $\geq 18 \log m$ which is the number of pieces into which each data item is split before encoding it with RS codes
$k$	set to $O(\log n)$ , system uses $k$ -ary butterfly as underlying topology
$\gamma$	set to $1/72$ and used in the term $\gamma n^{1/\log \log n}$ that denotes the maximum number of crashed servers allowed
$p$	positive constant in term $p \log n$ which is the length of an address
$\Lambda$	set to $p \log n$ which is the length of an address
$\gamma n^{1/\log \log n}$	upper bound for the number of crashed servers the system can tolerate
$\log_k n$	depth of the underlying $k$ -ary butterfly
$c/3$	number of pieces of a data item needed to recover that data item
$\beta c k^2$	maximum congestion at each intact server in each round of the decoding stage
$5c/6$	number of pieces of a requested data item to proceed with in the decodings tage

Table 1: Variables and terms used in this work and their meanings